UC Irvine
ICS Technical Reports

Title
Lessons learned from an application of static concurrency analysis

Permalink
https://escholarship.org/uc/item/03b358dt

Authors
Levine, David L.
Taylor, Richard N.

Publication Date
1991-10-30

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

N
699
C3
no. 91-72

Lessons Learned from an Application of Static Concurrency Analysis

David L. Levine
Richard N. Taylor
levine@ics.uci.edu and taylor@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine* 92717

October 30, 1991

UCI-ICS Technical Report 91-72
Arcadia Technical Report UCI-91-16

1 Introduction

Static concurrency analysis may be used to detect synchronization anomalies such as deadlocks and race conditions in concurrent programs [Tay83b]. Its notable drawback is that combinatorial state space explosion places a practical limit on the size of programs that may be analyzed [Tay83a]. One approach to assessing feasibility is to attempt to analyze representative concurrent programs. This report summarizes the results of one such exercise. We feel that the lessons learned will be valuable to those exploring similar paths.

The following section presents a brief outline of the foundation of our approach to static concurrency analysis; Section 3 contains more details of the toolset used. Sec-

*This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

tion 4 outlines the example program analyzed. The analysis results are summarized in Section 5 and Section 6 offers suggestions and directions for future efforts.

2 Concurrency Graphs

Our static analysis approach relies on *concurrency states*. In addition to being a basic concept in static analysis of concurrent programs [Apt83, Tay83b], this notion finds use in dynamic monitoring for errors in concurrent programming [GHL82] and in reachability analysis of Petri nets [Pet77].

The analysis approach is based on a graph model: each program unit, e.g., an Ada subprogram, task, package, or generic, defines a flowgraph. Every executable statement is represented by a flowgraph node and each transfer of program control corresponds to a directed edge. Flowgraph paths therefore represent statement sequences. Not all such statement sequences are executable, though, because control flow may be based on a possibly limited set of data values. Techniques such as symbolic execution and formal verification are used to determine path feasibility under all conditions.

Informally, a concurrency state embodies the next synchronization-related activity to occur in each task of a system. A state, therefore, is a tuple of nodes from the flowgraphs corresponding to each task. A concurrency history is a contiguous, legal sequence of concurrency states. The set of all such histories is the concurrency graph for a program.

The particular representation of concurrency graph we use is the Task Interaction Concurrency Graph (TICG) developed by Long and Clarke [LC89]. Instead of being based on control flow, it is derived from a flowgraph based on task interactions, called a Task Interaction Graph (TIG). The algorithm for constructing the TICG is straightforward: based on possible interactions as defined by the task TIGs, start from an initial state and add states adjacent to those already added to the TICG. The TICG is smaller (usually; more precisely, it is no larger) than a concurrency graph based on other flowgraph models. A further advantage is that

maximal sequential regions are identified in TIGs, facilitating analysis using appropriate techniques.

3 CATs

The Concurrency Analysis Tool Suite (CATs) was developed to automate the processes of building the TICG and its related histories [YTFB89]. CATs comprises a variety of small tool components as shown in Figure 1, taken from [You89], and supports a variety of analyses. We utilized it to construct the TICG and then statically check for deadlock and for violations of temporal logic assertions (TLAs). The figure shows tools as boxes and data between horizontal line pairs. The only operational compiler front end at present is for Ada source. The remainder of this paper uses Ada terminology for convenience, although in principle CATs can be used to analyze concurrent programs written in other languages.

The compiler front end factors out the source code manipulations common to other tools and builds abstract syntax graphs. TLAs, described briefly below, are embedded in the source as annotations and extracted by the special purpose assertion extractor. The temporal logic model checker performs, in addition to TLA verification, the deadlock checking because it is considered an implicit specification for all concurrent programs. If a possible deadlock or TLA violation is detected, the report contains two components: an example trace of task interaction events leading to the violation and a snapshot of the concurrency state listing the next tasking-related activities to be performed by each task. Since we were focussing on static analysis, the symbolic execution component, ARIES, was not exercised.

The tool for constructing task interaction graphs, TIGGER, is not completed so the TIGs were constructed by hand. In addition, the compiler front end does not yet handle languages other than Ada so it is not possible to trace task interactions through, for example, the C++ code in the Chiron system that we analyzed (introduced in Section 4). An important optimization was performed to ease the labor-intensive TIG-building task, namely task activations and terminations were

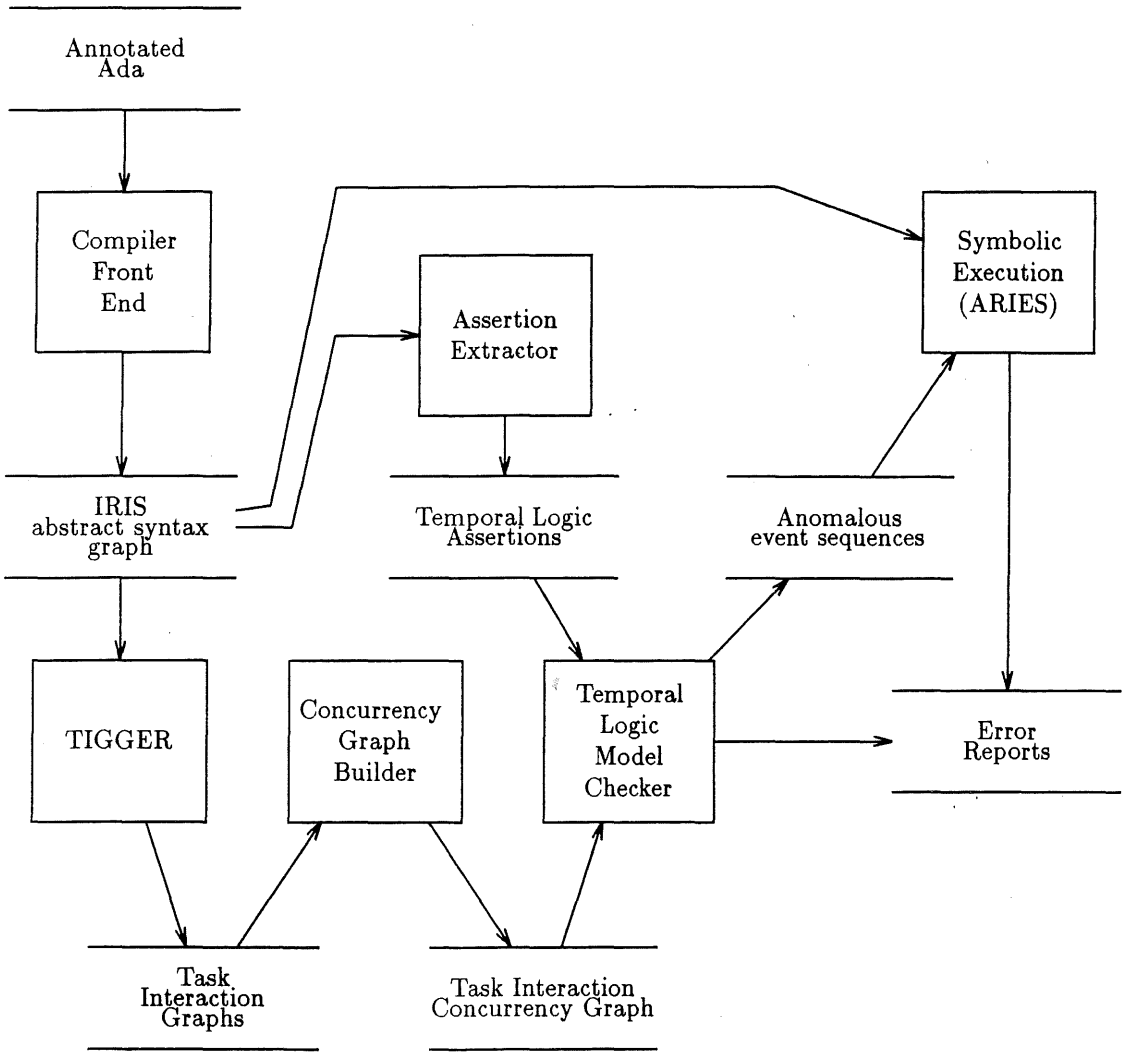

Figure 1: CATs Components and Information Flow [You89]

not modeled. There were no task entry calls during initialization and no task interactions dependent on specific task activation interleavings in the example Chiron 1.0 program, so activations were not of particular interest. Task terminations only occur when the entire program is shut down, but we were only interested in the program as it was running prior to shutdown. These would have increased the size of the TICGs without adding much to the understanding of this example, although in general they may very well need to be considered.

Deadlock and TLAs. A deadlock state is defined as one in which no task may progress. Partial deadlock situations in which several tasks can never proceed but at least one pair can rendezvous[1] are not identified by the deadlock checker. A single rendezvous that can occur at any time could hide what otherwise would be a deadlock state. If such situations are of interest, the analyst can isolate them by embedding TLAs that assert some eventual task progression.

TLAs are expressed in a propositional, branching time logic based on the computation tree logic of Clarke, Emerson, and Sistla [CES86]. The flavor our approach can be conveyed by looking at its components. TICG edges, which represent tasking events such as rendezvous acceptance or completion, correspond to events in the logic. Atomic propositions are combinations of the temporal logic operations of *eventually*, *always*, *potentially*, and *until* on TICG nodes. The usual boolean operations may be used to combine atomic propositions and event descriptions to create assertions.

4 Chiron

The Chiron 1.0 user interface development system [KCTT91] was selected for our static concurrency analysis because it represents a modern, fairly large system[2],

[1]Technically, if one task can progress without performing a task interaction, e.g., Milner's silent step τ [Mil80], there would not be deadlock. However, such sequential activity would be hidden in a single TICG node and would not be visible to the deadlock checker.

[2]Chiron comprises roughly 10^5 source lines of code, designed and implemented over a two year period with the equivalent of five individuals working at any one time. It is based on a concurrent

because it includes a reasonably complex task structure, and because it was not "designed to be analyzed" (it is not a toy example). Furthermore, the implementation includes two languages, Ada and C++. The user interface developer builds one or more artists for the application tool; a key feature is that the tool is viewed as an abstract data type (ADT), with all access controlled by a dispatcher. Artists are notified of changes of state in the tool, and may alter the view presented to the user(s) as appropriate. Artists may also respond directly to events initiated by the user(s), and inform the tool of necessary changes through the dispatcher.

Chiron provides the artist writer with an extensible C++ hierarchy of graphical objects, termed the Abstract Depiction Language (ADL) hierarchy. While most of the code of interest in the concurrency analysis is Ada, calls from Ada tasks through the ADL hierarchy must be followed in order to determine all possible interactions.

The task and package structure is shown in Figure 2 using a variation of Buhr diagrams. Rectangles depict packages, nested as drawn. Parallelograms represent tasks, with arrows from callers to entries. Shadowed parallelograms with dashed lines indicate optional multiple task instances. Curved-back arrows represent guarded entry calls. While this figure is intended to highlight task interactions, the ADT is not a task so the arrows leading to it are simply procedure calls.

The particular task structure shown in Figure 2 is for the Dialogue demonstration application, which highlights the ADT-based nature of the Chiron approach by providing an interface for interactive manipulation of a stack. The only application-specific portion of the task structure is the number of artists. The simplest configuration, chosen for our analysis, has just one.

All ADT operations, namely stack operations such as create, push, and pop, are performed by either the main task, labeled Start_Tool, or by the artist in response to user requests (events). The artist has a separate task entry for each operation. Because there is no distinction between these various operations from a task interaction viewpoint, they are lumped together as a single "artist actions" entry in our model. Access to the ADT by Start_Tool and the artist is serialized by way

control model with the user interface and tool simultaneously active.

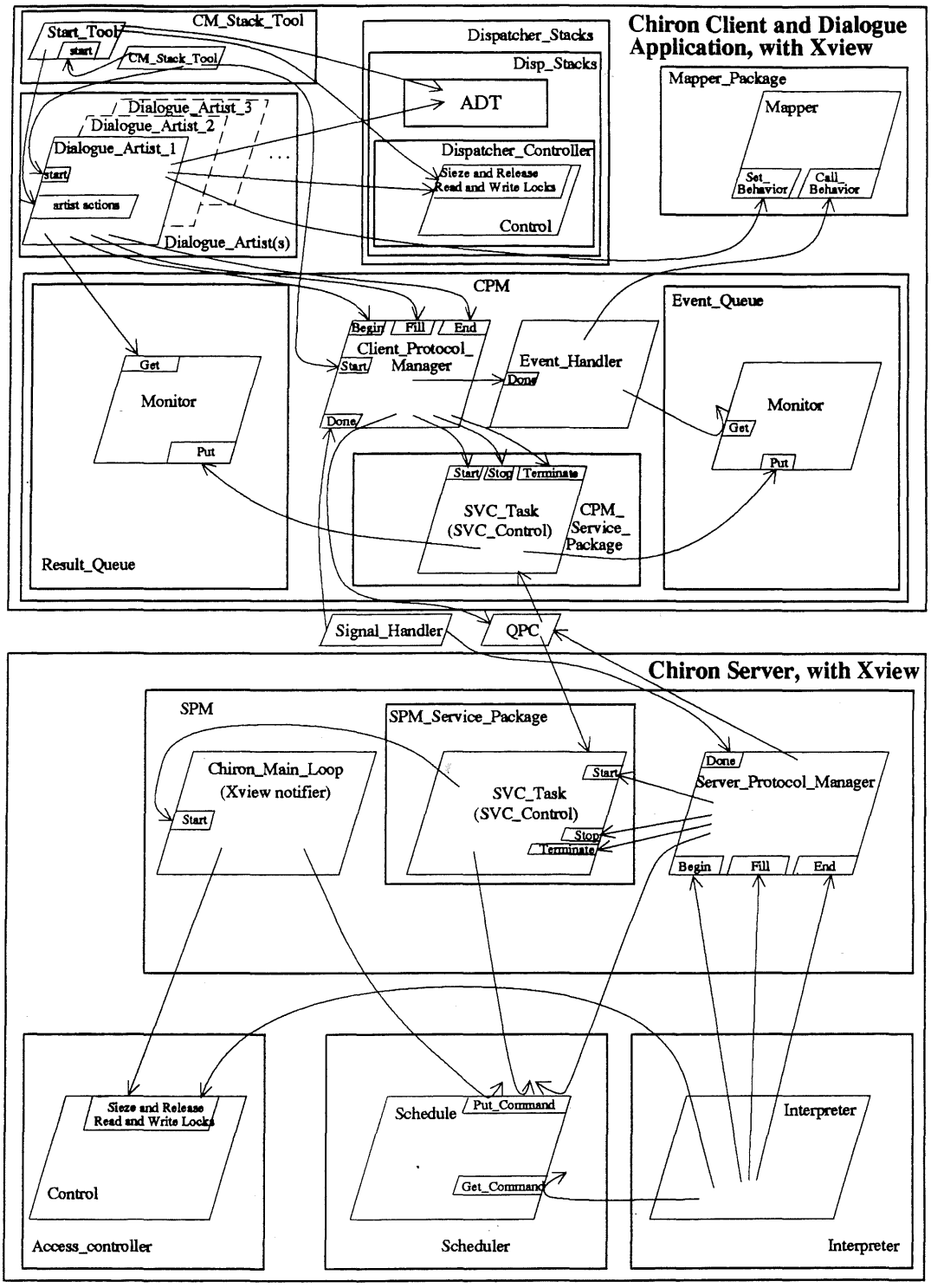

Figure 2: Chiron 1.0 Task and Package Structure

of a conventional readers/writers lock mechanism, implemented as the Control task in the Dispatcher_Controller package. A two-level package interface to the Control task handles the rather intricate dispatcher details such as notifying all other artists of a change to the ADT by any one artist or the main task.

The Chiron design separates it into two operating system processes, the server and the client/tool. Our analysis was partitioned, or parceled, along this split. The two "in-between" tasks provide a higher level interface for handling operating system signals and all interprocess communication. The relevant portions of these tasks were included in the analysis on each side of the split.

Analysis Parceling. CATs does not yet provide any guidance for parceling so it is performed manually. A mechanical process for parceling based on biconnected components of the TICG is described in [Tay83b], but requires that certain properties of the components be met. If this is not the case, or if the biconnected components are still too large, the analyst must guide the parceling. Young describes one such approach based on *weak monitors* [You89]. These assure mutual exclusion to any data item protected by the monitor and permit multiple simultaneous calls on access tasks and procedures, on condition that caller activities not affect the termination of rendezvous in progress.

A good system design emphasizes loose coupling between components, so natural system boundaries delineate parcels most likely to have minimum interactions. Although parcels are analyzed independently, interactions between them must be considered. An interaction is a rendezvous with a task in another parcel. One characteristic to consider when checking for deadlocks involving interactions that cross parcel boundaries is whether the rendezvous can occur repeatedly. If so, they could hide what would otherwise be deadlock in the parcel. A mechanical method for determining whether a single interaction could hide deadlock is to disable the interaction and perform the deadlock check. If deadlock is reported only with the interaction disabled, it may hide deadlock.

Table I: TICG Size and Construction/Deadlock Check Times

Component	*Tasks*	*States*	*Edges*	*Time, sec.*
server	8	9494	35612	45.2
client/application	12	22993	99497	113.9

5 Analysis Results

A primary goal of the analysis was to assess the feasibility of static concurrency analysis for a program of representative size and task interaction complexity. The observables were:

- TICG size and construction times for both the server and client/tool
- number of potential deadlock states, and whether each is feasible
- whether there were any other tasking anomalies
- an indication of the utility of temporal logic assertions

TICG Size. As shown in Table I, the concurrency state graphs are of a very manageable size.[3] A noteworthy lesson is that static concurrency analysis indeed appears to be practical for representative systems. Furthermore, if the program is too large to be handled in its entirety, parcels of about 12 tasks should be manageable.

Roughly one-third of the execution time was spent building the TICGs and two-thirds checking for deadlock. This ratio is specific to the particular TICGs analyzed. The deadlock check algorithm is NP-complete: the check of each concurrency state at worst may require time exponential in the number of TIG nodes. However, many states may in fact be checked very quickly, and information may be shared between

[3]The approximate execution times in Table I are for concurrency state graph construction on a Sun 4/490 with Verdix Ada 6.0.3. A Sun SparcStation 2 offers similar performance.

checks of other states. These results show that the NP-complete result need not preclude efficient performance in practice.

Deadlock States. The deadlock check reported 24 potential deadlocks on the server side. Further investigation revealed that none are in fact feasible. The following summary of that investigation highlights the subtleties of static deadlock checking and sets the stage for discussing the true deadlock found on the client/tool side.

The server has a graceful shutdown mechanism which is initiated by the Signal_Handler task after catching a user-originated kill signal from the operating system. Of the potential deadlocks, 22 were after the Signal_Handler has started the shutdown. This is not unexpected because we don't model `terminate` and some tasks wait on that alternative. To verify the shutdown hypothesis, the Signal_Handler was disabled by disconnecting from its TIG initial node the edge representing the shutdown rendezvous. The 22 previously reported shutdown deadlocks vanished, confirming their origin.

One of the two remaining potential deadlock reports is shown below:

```
***Potential deadlock after this sequence:
    Synchronize server_protocol_manager, spm_svc_task.svc_start
    Engage qpc_call, spm_svc_task.message
    Finish qpc_call, spm_svc_task.message
    Synchronize spm_svc_task, chiron_main_loop.start
    Engage chiron_main_loop, access_control.start
    Finish chiron_main_loop, access_control.start
    Engage qpc_call, spm_svc_task.message
    Finish qpc_call, spm_svc_task.message
state  2027:
access_control attempting to accept interpreter.stop_write
access_control attempting to accept interpreter.write
access_control attempting to accept interpreter.stop_read
access_control attempting to accept interpreter.start
access_control attempting to accept chiron_main_loop.stop_write
access_control attempting to accept chiron_main_loop.write
access_control attempting to accept chiron_main_loop.stop_read
access_control attempting to accept chiron_main_loop.start
chiron_main_loop attempting to engage access_control.write
interpreter is attempting to engage access_control.start
schedule attempting to accept interpreter.get_command_result
schedule attempting to accept interpreter.get_command_halt
schedule attempting to accept interpreter.get_command_event
```

```
schedule attempting to accept interpreter.get_command_instruction
schedule attempting to accept spm_svc_task.put_command
schedule attempting to accept server_protocol_manager.put_command
schedule attempting to accept chiron_main_loop.put_command
server_protocol_manager attempting to synchronize with signal_handler.done
server_protocol_manager attempting to accept interpreter.end_message
server_protocol_manager attempting to accept interpreter.fill_message_artist
server_protocol_manager attempting to accept interpreter.begin_message_artist
server_protocol_manager attempting to accept interpreter.fill_message_instruc
server_protocol_manager attempting to accept interpreter.begin_message_instruc
spm_svc_task is attempting to synchronize with chiron_main_loop.start
signal_handler is not attempting to move.
qpc_call attempting to engage spm_svc_task.message
qpc_call attempting to accept server_protocol_manager.message
```

The report begins with one of possibly many example task interaction traces that lead to the potential deadlock state. The state itself is then described in terms of the next task interaction that could be performed by each task. More than one interaction alternative indicates a `select` statement. The number assigned to the state, 2027 in this case, has no semantic value other than as a unique state identifier. The other potential deadlock is very similar to this one; the only difference being that the Interpreter had completed the rendezvous with Access_Control.start and was attempting to engage the write entry.

Both potential deadlock reports were due to the pessimistic inaccuracy introduced by data folding. The Access_Controller is a simple implementation of the multiple readers/single writer problem. Guarded **accept** statements prohibit write access if there are any readers and read access if there is a writer. Static analysis can not distinguish data values for the guards, including the case when all guards are closed. Potential deadlock is reported when deadlock could occur with any possible assignment of data values, hence the description, pessimistic. In state 2027 above, both the Chiron_Main_Loop and Interpreter tasks are attempting to engage Access_Controller entries, which the Access_Controller is reported as ready to accept. Not shown in the report is the fact that the Access_Controller entries are guarded. Inspection reveals that the program logic does not allow any set of data values such that all the guards are closed, and therefore one of the Access_Controller rendezvous would in fact be completed.

On the client/tool side, there were 17 reported potential deadlocks. Fifteen were

due to the same shutdown mechanism as on the server side. The remaining two represent actual deadlock states, and have been observed in the implementation. One is shown below, the other is similar: instead of the Dispatcher_Control waiting on the Start_Tool.Stop_Write entry, it waits on Dialogue_Artist_1.Stop_Write. An example sequence of synchronization events leading to this state is shown below in the discussion of TLAs.

```
state 850:
client_protocol_manager attempting to synchronize with signal_handler.done
client_protocol_manager attempting to accept dialogue_artist_1.end_message
client_protocol_manager attempting to accept dialogue_artist_1.fill_message
client_protocol_manager attempting to accept dialogue_artist_1.begin_message
cm_stack_tool is not attempting to move.
cpm_svc_task attempting to synchronize with client_protocol_manager.svc_stop
cpm_svc_task attempting to engage result_queue_monitor.put
cpm_svc_task attempting to engage event_queue_monitor.put
dialogue_artist_1 attempting to engage dispatcher_control.set_write_lock
dispatcher_control attempting to accept start_tool.stop_write
event_handler attempting to synchronize with client_protocol_manager.done
event_handler attempting to engage event_queue_monitor.get
event_queue_monitor attempting to accept event_handler.get
event_queue_monitor attempting to accept cpm_svc_task.put
mapper attempting to accept event_handler.call_behavior
mapper attempting to accept dialogue_artist_1.set_behavior
result_queue_monitor attempting to accept dialogue_artist_1.get
result_queue_monitor attempting to accept cpm_svc_task.put
start_tool attempting to engage dialogue_artist_1.artist_action
signal_handler is not attempting to move.
qpc_call attempting to synchronize with client_protocol_manager.quit
qpc_call attempting to engage cpm_svc_task.message
qpc_call attempting to accept client_protocol_manager.message
```

Deadlock is caused by the circular wait between Dialogue_Artist_1, Dispatcher_Control, and Start_Tool. The tool, after performing a write to the ADT, is trying to send update information to the artist, which is waiting for a write lock from the dispatcher. The artist is responding to an event such as a button press sent by the server, and waiting for a write lock from the dispatcher.

There were two instances of analyst intervention on the client/tool side, one required for realistic analysis and one optional. First, when dividing the tasks into parcels, interactions across parcel boundaries were checked for any rendezvous that could hide otherwise deadlocked states in the parcels. One was recognized in the client/tool parcel: the QPC can call a client side SVC_Task entry at will,

representing a event being passed in from the server. Strictly speaking, deadlock is not possible in the client/tool parcel when considered in isolation (or in the system as a whole), because a modeled user event can occur at any time. An otherwise deadlocked state could have been hidden from our analysis simply because external events could continue to occur. Therefore, the QPC/SVC_Task rendezvous was disabled. To verify the suspicion that deadlocks could be hidden, we enabled the rendezvous: no deadlocks were reported.

The second intervention into the analysis was optional but obviated the need to search beyond the information conveyed in the deadlock report. The first attempt at analysis on the client/tool side was with a version of the Dispatcher_Controller Control task that was structurally similar to the Access_Controller on the server side. That is, the task consisted of a `select` statement with `accept` alternatives for each of the various read and write lock entries, some guarded. While the true deadlock state was found, it was not clearly distinguishable from those reported deadlock states that were not feasible due to permitted guard values. The analysis results displayed above are for a version of the Control task with unrolled loops, effectively replacing the guards with loop structures. The cause of deadlock is then evident from the above report, which does not include any information about guards.

Other Tasking Anomalies. While constructing the TIGs for each of the tasks, two synchronization anomalies were noted. One involved a race condition on an unprotected variable in the Dispatcher_Controller, a variable writable both by the Control task and by others outside the package through access procedures. This anomaly could have been modeled in the TIGs by considering all accesses of the shared variable to be task interactions. However, it was decided to repair the implementation by serializing all access through the Control task and carry the analysis forward.

The other anomaly involved a race condition on a variable that keeps track of the number of artists that are processing dispatcher calls. This does not affect our analysis because there is only one artist. Nonetheless, the dispatcher is being

redesigned to eliminate this anomaly.

TLAs. The utility of TLAs is demonstrated by way of example. The simple TLA

```
eventually finish start_tool.artist_action
```

can not be true if the client/tool deadlock state shown above is ever reached. The model checker verifies this, and presents an example sequence of events which violate the assertion. The "<<STUCK>>" message indicates that a deadlock state has been reached.

```
***Violation of: (eventually finish start_tool.artist_action)
    Synchronize cm_stack_tool, client_protocol_manager.start
    Synchronize cm_stack_tool, dialogue_artist_1.start
    Synchronize cm_stack_tool, start_tool.start
    Engage dialogue_artist_1, mapper.set_behavior
    Finish dialogue_artist_1, mapper.set_behavior
    Engage dialogue_artist_1, dispatcher_control.set_write_lock
    Finish dialogue_artist_1, dispatcher_control.set_write_lock
    Engage dialogue_artist_1, dispatcher_control.start_read
    Finish dialogue_artist_1, dispatcher_control.start_read
    Engage dialogue_artist_1, dispatcher_control.stop_read
    Finish dialogue_artist_1, dispatcher_control.stop_read
    Engage start_tool, dispatcher_control.set_write_lock
    Finish start_tool, dispatcher_control.set_write_lock
    Engage dialogue_artist_1, dispatcher_control.unset_write_lock
    Finish dialogue_artist_1, dispatcher_control.unset_write_lock
    Engage start_tool, dispatcher_control.start_write
    Finish start_tool, dispatcher_control.start_write
    Engage start_tool, dispatcher_control.write
    Finish start_tool, dispatcher_control.write
    Synchronize client_protocol_manager, qpc_call.register
    Synchronize client_protocol_manager, cpm_svc_task.svc_start
    Synchronize client_protocol_manager, qpc_call.link
    Engage dialogue_artist_1, client_protocol_manager.begin_message
    Finish dialogue_artist_1, client_protocol_manager.begin_message
    Engage dialogue_artist_1, client_protocol_manager.fill_message
    Finish dialogue_artist_1, client_protocol_manager.fill_message
    Engage dialogue_artist_1, client_protocol_manager.end_message
    Finish dialogue_artist_1, client_protocol_manager.end_message
    Engage client_protocol_manager, qpc_call.message
    Finish client_protocol_manager, qpc_call.message
    Engage dialogue_artist_1, result_queue_monitor.get
    Finish dialogue_artist_1, result_queue_monitor.get
    Engage dialogue_artist_1, mapper.set_behavior
    Finish dialogue_artist_1, mapper.set_behavior
<<LOOP>>
<<STUCK>>
```

TLAs are evaluated in some context, typically the executable statement preceeding the appearance of the TLA in the source code. This is conveyed to the TICG by associating every TICG state that includes the TIG node corresponding to that statement with the context of TLA.

Analyst intervention is often required with the present implementation of the model checker because it does not assume a fair task scheduler. Therefore, as with the deadlock checker, task interactions that can occur at any time may mask those of interest. In particular, a sequence of interactions may cycle forever without fair scheduling. These are clearly identified in the TLA sample sequence report. To avoid this situation in the Chiron client/tool side analysis, two tasks were partially disabled in the same manner as for the deadlock check: the QPC task and the Event_Handler task. This modeling limitation is not fundamental: the temporal logic model checking algorithm can be adapted to handle fairness [CES86].

Selection of TLAs requires familiarity with the (expected) operation of the system, and is therefore best performed by the system designer. Therefore, TLAs can be used as behavioral specifications to which the design and implementation are verified.

6 Lessons Learned

Several other lessons were learned during this exercise. They involve interlanguage analysis, deadlock checking, and automation of the analysis process.

Interlanguage analysis should be feasible if the underlying tasking models are compatible. We did not explore this in depth because nearly all of the Chiron task interactions involved Ada tasks. The exceptions are the Xview notifier and operating system signals. We satisfactorily modeled both as simple event generators. The several Ada task entry calls in the C++ hierarchy were easy to identify because the interlanguage interfaces are explicit in the code.

A final lesson is an important one, but is not evident from the final results: automation removes a source of subjectivity in the analysis. The TIGs were con-

structed by hand, and not unexpectedly, errors crept in. The first error was to not assign edge groups, used by the deadlock checker to identify the alternatives of `select` statements. This did not affect the composition of the TICGs, but many deadlock states were erroneously reported. The second error involved several simple mistakes in manually tracing task interactions through the source code.

Summary. The lessons learned are summarized as follows.

- Static concurrency analysis may be feasible for systems of reasonable size and complexity.
- If the analysis must be divided, parcels of 12 tasks are manageable.
- Interactions across parcel boundaries demand careful attention.
- Interlanguage analysis is feasible with compatible tasking models.
- The deadlock checking algorithm, though NP-complete, offers good performance for typical programs.
- The deadlock checker does not identify starved tasks, or partial deadlock among a task subset.
- Temporal logic assertions are a useful analysis tool, but may be most beneficial when inserted by the designer.
- Each step in the analysis process should be automated.

Our future efforts will be directed at finishing TIGGER, adapting the temporal logic model checker to handle fairness, and developing a user-friendly front end to integrate the various CATs components.

References

[Apt83] Krzysztof R. Apt. A static analysis of CSP programs. In *Proceedings of the Workshop on Program Logic*, Pittsburgh, PA, June 1983.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[GHL82] Steven M. German, David P. Helmbold, and David C. Luckham. Monitoring for deadlocks in Ada tasking. In *Proceedings of the AdaTEC Conference on Ada*, pages 10–25, Arlington, VA, October 1982.

[KCTT91] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[LC89] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the Eleventh International Conference on Software Engineering*, Pittsburgh, May 1989.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1980.

[Pet77] James L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[Tay83a] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[You89] Michal Young. *Hybrid Analysis Techniques for Software Fault Detection.* PhD thesis, University of California, Irvine, 1989. Available as UCI ICS technical report 89–26.

[YTFB89] Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 200–209, Key West, Florida, December 1989. Published as *ACM SIGSOFT Software Engineering Notes 14*(8).